1_Tokenizacion

February 7, 2025

Tema 1: Computación Inteligente (LLM)

Tokenización

Prof. Wladimir Rodríguez

wladimir@ula.ve

Departamento de Computación

0.1 ¿Qué es un Tokenizador?

- **Definición**: Un tokenizador es un componente esencial en el procesamiento del lenguaje natural (PNL) que:
  - Divide el texto de entrada en unidades más pequeñas llamadas "tokens".
  - Estos tokens pueden ser:
    - Palabras
    - Subpalabras (fragmentos de palabras)
    - Caracteres

0.2 La Importancia de la Tokenización

- **Representación Numérica**: Los LLMs son modelos numéricos. Necesitan convertir el texto en números (IDs de tokens) para poder procesarlo. Sin tokenización, no hay entrada para el modelo.
- **Definición del Vocabulario**: El tokenizador define el vocabulario del modelo: el conjunto de tokens que el modelo "conoce". Esto afecta directamente a la capacidad del modelo para representar el lenguaje.
- **Rendimiento**: La eficiencia del tokenizador impacta la velocidad de entrenamiento e inferencia del LLM. Un tokenizador lento puede convertirse en un cuello de botella.
- **Comprensión**: La forma en que se tokeniza el texto influye en la capacidad del modelo para comprender el significado. Un buen tokenizador ayuda al modelo a capturar las relaciones semánticas.

0.3 Tokenización Basada en Palabras

Ejemplo:

- Texto: "¡Hola mundo! ¿Cómo estás?"

- Tokens (Basado en Espacios): ["¡Hola", "mundo!", "¿Cómo", "estás?"]

Problemas:

- *Puntuación*: La puntuación está pegada a las palabras, dificultando el aprendizaje de relaciones sintácticas y semánticas.
- *Contracciones*:
  - She's (Ella es/está): Esta es una contracción de "She is" o "She has".
  - isn't (no es/está): Esta es una contracción de "is not".
- *Idiomas*: ¿Cómo manejar idiomas sin espacios entre palabras (chino, japonés)?

0.3.1 Cargar la novela Doña Barbara

Cargar el texto sin formato con el que queremos trabajar

[2]:
```
with open("../Datos/DonaBarbara.txt", "r", encoding="utf-8") as f:
    texto_sin_formato = f.read()

print("Número total de caracteres:", len(texto_sin_formato))
print(texto_sin_formato[:99])
```

```
Número total de caracteres: 596622
Doña Bárbara
(Caracas: Editorial Araluce, 1929, 480 págs.)



PRIMERA PARTE


I
¿Con quién vamos?
```

- El objetivo es tokenizar y vectorizar (embed) este texto para un LLM
- Desarrollemos un tokenizador simple basado en un texto de muestra simple que luego podamos aplicar al texto anterior

[4]:
```
import re
texto = 'Hola mundo. Esto, es una prueba'
resultado = re.split(r'(\s)', texto)
print(resultado)
```

```
['Hola', ' ', 'mundo.', ' ', 'Esto,', ' ', 'es', ' ', 'una', ' ', 'prueba']
```

No solo queremos dividir en espacios en blanco, sino también en comas y puntos, así que modifiquemos la expresión regular para hacer eso también.

[5]:
```
resultado = re.split(r'([,.]|\s)', texto)
print(resultado)
```

```
['Hola', ' ', 'mundo', '.', '', ' ', 'Esto', ',', '', ' ', 'es', ' ', 'una', '
', 'prueba']
```

Como podemos ver, esto crea cadenas vacías, eliminémoslas.

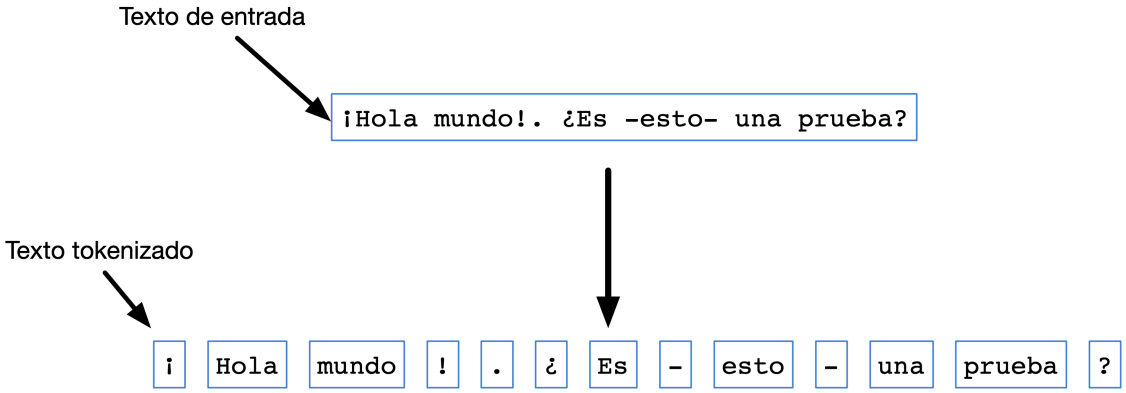
[6]:
```
resultado = [item for item in resultado if item.strip()]
print(resultado)
```

```
['Hola', 'mundo', '.', 'Esto', ',', 'es', 'una', 'prueba']
```

Esto se ve bastante bien, pero también manejaremos otros tipos de puntuación, como punto y coma, signos de interrogación, etc.

[7]:
```
texto = '¡Hola mundo!. ¿Es -esto- una prueba?'
resultado = re.split(r'([,.:;¿?_!¡"()\']|-|\s)', texto)
resultado = [item.strip() for item in resultado if item.strip()]
print(resultado)
```

```
['¡', 'Hola', 'mundo', '!', '.', '¿', 'Es', '-', 'esto', '-', 'una', 'prueba',
'?']
```



0.3.2 Ventajas

- **Intuición:** Representa unidades de significado relativamente claras. Las palabras generalmente tienen un significado semántico discernible.
- **Secuencias más cortas:** En comparación con la tokenización basada en caracteres, genera secuencias más cortas, lo que reduce la carga computacional.
- **Fácil de entender e interpretar:** Los tokens son fácilmente comprensibles para los humanos.

0.3.3 Desventajas

- **Tamaño de vocabulario grande:** El número de palabras únicas en un idioma puede ser muy grande, lo que resulta en un vocabulario grande que requiere más memoria y puede hacer que el entrenamiento del modelo sea más difícil.
- **Problema de palabras fuera del vocabulario (OOV):** El modelo no puede procesar palabras que no están en su vocabulario. Esto puede ser un problema si el texto contiene palabras raras, nombres propios, errores ortográficos o palabras nuevas.
- **Dificultad para manejar inflexiones:** Las diferentes formas de una misma palabra (e.g., "correr", "corriendo", "corrió") se tratan como palabras diferentes, lo que dificulta que el modelo generalice entre ellas.
- **Manejo inconsistente de la puntuación:** La forma en que se maneja la puntuación puede variar según el tokenizador, lo que puede afectar la calidad del modelo.

[8]:
```
print(sorted(list(set(resultado))))
```

```
['!', '-', '.', '?', 'Es', 'Hola', 'esto', 'mundo', 'prueba', 'una', '¡', '¿']
```

[10]:
```
preprocesado = re.split(r'([,.:;¿?_!¡"()\']|-|\s)', texto_sin_formato)
preprocesado = [item.strip() for item in preprocesado if item.strip()]
print(preprocesado[:30])
```

```
['Doña', 'Bárbara', '(', 'Caracas', ':', 'Editorial', 'Araluce', ',', '1929',
',', '480', 'págs', '.', ')', 'PRIMERA', 'PARTE', 'I', '¿', 'Con', 'quién',
'vamos', '?', 'Un', 'bongo', 'remonta', 'el', 'Arauca', 'bordeando', 'las',
'barrancas']
```

Calculemos el número total de tokens

[12]:
```
print(len(preprocesado))
```

```
115979
```

0.3.4 Tokenización Basada en Caracteres

- **Descripción:**
  - El enfoque más granular: cada carácter individual en el texto se convierte en un token.
  - El vocabulario consiste en el conjunto de caracteres únicos presentes en los datos de entrenamiento.
  - Espacios en blanco, puntuación y otros símbolos se tratan como caracteres regulares.
- **Ejemplo:**
  - Texto: "Hola mundo!"
  - Tokens: ["H", "o", "l", "a", " ","m", "u", "n", "d", "o", "!"]

0.3.5 Ventajas

- **Tamaño de vocabulario extremadamente pequeño**: Ideal para idiomas con conjuntos de caracteres grandes o cuando se busca minimizar la huella de memoria del modelo. El tamaño del vocabulario es limitado al conjunto de carácteres usados
- **Robustez frente a palabras fuera del vocabulario (OOV):** No hay problema de OOV, ya que cualquier texto puede ser representado como una secuencia de caracteres conocidos.
- **Simple de implementar:** La lógica de tokenización es trivial.
- **Útil para manejar errores ortográficos:** Modelos basados en caracteres pueden ser más robustos ante errores ortográficos o variaciones en la escritura.

0.3.6 Desventajas

- **Secuencias largas:** Las oraciones se convierten en secuencias muy largas de tokens, lo que aumenta la carga computacional, especialmente para modelos basados en Transformers, donde la atención se calcula sobre todas las parejas de tokens.
- **Falta de significado a nivel de token:** Los caracteres individuales tienen poco significado semántico por sí mismos. El modelo debe aprender relaciones complejas entre caracteres para comprender el texto.
- **Dificultad para capturar dependencias a largo alcance:** Debido a las secuencias largas, puede ser difícil para el modelo aprender dependencias entre palabras que están separadas por muchos caracteres.
- **Ineficiencia en la representación:** Se requiere más "esfuerzo" para el modelo aprender el significado en comparación con la tokenización a nivel de palabra o subpalabra.

[11]:
```
caracteres = sorted(list(set(texto_sin_formato)))
tamaño_vocabulario = len(caracteres)
print(''.join(caracteres))
print(tamaño_vocabulario)
```

```
 !()*,-
.01234589:;?ABCDEFGHIJLMNOPQRSTUVXYZabcdefghijklmnopqrstuvwxyz¡¿ÁÉÑÓÚáéíñóúü-’""
…
89
```

0.3.7 Tokenización Basada en Subpalabras:

- **Motivación:**
  - Superar las limitaciones de la tokenización basada en palabras y caracteres.
  - Reducir el tamaño del vocabulario en comparación con la tokenización basada en palabras.
  - Manejar palabras fuera del vocabulario (OOV) de forma más eficaz.
  - Capturar la estructura morfológica de las palabras

- **Descripción:**
  - Divide las palabras en unidades más pequeñas llamadas "subpalabras".
  - Las subpalabras pueden ser morfemas (unidades significativas más pequeñas), fragmentos de palabras o caracteres individuales.
  - El vocabulario consiste en un conjunto de subpalabras que se aprenden a partir de los datos de entrenamiento.

0.3.8 Algoritmos Comunes de Tokenización de Subpalabras

- **Byte Pair Encoding (BPE):**
  - *Proceso*:
    1. Comienza con un vocabulario que contiene todos los caracteres individuales.
    2. Iterativamente, identifica el par de tokens más frecuente en los datos de entrenamiento y los fusiona en un nuevo token.
    3. Repite el paso 2 hasta que se alcance el tamaño de vocabulario deseado.
  - *Ejemplo*:
    * Datos: "low", "lower", "lowest"
    * BPE podría fusionar "l" y "o" -> "lo", luego "lo" y "w" -> "low", etc.
  - *Ventajas*: Sencillo de implementar, eficaz para reducir el tamaño del vocabulario.
  - *Desventajas*: Puede crear subpalabras que no tienen un significado claro.
- **WordPiece:**
  - *Proceso*:
    1. Comienza con un vocabulario que contiene todos los caracteres individuales.
    2. Iterativamente, identifica el par de tokens que maximiza la probabilidad de los datos al ser fusionados en un nuevo token. Usa un modelo de lenguaje para evaluar la probabilidad.
    3. Repite el paso 2 hasta que se alcance el tamaño de vocabulario deseado.
  - *Diferencia clave con BPE*: En lugar de solo la frecuencia, WordPiece usa una métrica de verosimilitud basada en el modelo de lenguaje.
  - *Ventajas*: Similar a BPE, pero con una base probabilística más sólida.
  - *Desventajas*: Similar a BPE.
  - *Usado por*: BERT, DistilBERT, MobileBERT.
- **Unigram Language Model:**
  - Proceso:
    1. Comienza con un vocabulario grande (e.g., todas las palabras y caracteres).

2. Asigna una probabilidad a cada token en el vocabulario.
3. Iterativamente, elimina los tokens que menos contribuyen a la probabilidad del corpus, hasta alcanzar el tamaño de vocabulario deseado.
4. Para tokenizar, se busca la segmentación del texto en tokens que maximice la probabilidad de la secuencia.

- *Ventajas*: Permite múltiples segmentaciones, útil para tareas como la segmentación de palabras en chino.
- *Desventajas*: Más complejo de implementar que BPE y WordPiece.
- *Usado por*: SentencePiece (puede usar Unigram).

- **SentencePiece:**
  - *Características*:
    * Es un tokenizador independiente del idioma que trata el texto como una secuencia de caracteres Unicode.
    * Maneja los espacios en blanco como caracteres regulares, lo que evita problemas con la tokenización de espacios en blanco.
    * Proporciona una interfaz unificada para BPE, WordPiece y Unigram.
  - *Ventajas*: Fácil de usar, versátil, ideal para modelos multilingües.
  - *Desventajas*: Puede ser un poco más lento que otros tokenizadores.
  - *Usado por*: T5, ALBERT, Marian, muchos modelos modernos.

0.3.9 Ventajas

- **Vocabulario moderado:** El tamaño del vocabulario es mucho menor que el de la tokenización basada en palabras, pero mayor que el de la tokenización basada en caracteres.
- **Manejo eficaz de OOV:** Puede representar palabras desconocidas como combinaciones de subpalabras conocidas.
- **Captura de la estructura morfológica:** Puede aprender a representar morfemas y otros fragmentos de palabras que tienen significado.
- **Generalización mejorada:** Permite que el modelo generalice mejor a palabras nuevas y raras.

0.3.10 Desventajas

- **Mayor complejidad:** Más complejo de implementar y entender que la tokenización basada en palabras o caracteres.
- **Tokens menos intuitivos:** Los tokens de subpalabras pueden ser menos intuitivos para los humanos que las palabras completas.
- **Posible pérdida de información:** En algunos casos, la división en subpalabras puede resultar en la pérdida de información semántica.

0.4 Crear un tokenizador

-

0.4.1 Convertir tokens en identificadores de token

– A continuación, convertimos los tokens de texto en identificadores de token que podemos procesar mediante capas de vectorización (embedding) más tarde.

[13]:
```
todas_las_palabras = sorted(set(preprocesado))
tamaño_vocabulario = len(todas_las_palabras)

print(tamaño_vocabulario)
```

```
14724
```

[14]:
```
vocabulario = {token:entero for entero,token in enumerate(todas_las_palabras)}
```

A continuación se muestran las primeras 50 entradas de este vocabulario:

[15]:
```
for i, item in enumerate(vocabulario.items()):
    print(item)
    if i >= 50:
        break
```

```
('!', 0)
('(', 1)
(')', 2)
('*', 3)
(',', 4)
('-', 5)
('.', 6)
('1929', 7)
('20', 8)
('33', 9)
('480', 10)
('5', 11)
('90', 12)
(':', 13)
(';', 14)
('?', 15)
('A', 16)
('Abajo', 17)
('Abandonarla', 18)
('Abandonó', 19)
('Abeja', 20)
('Abran', 21)
('Acababa', 22)
('Acabar', 23)
('Acabarían', 24)
```

```
('Acabe', 25)
('Acabó', 26)
('Acariciándolo', 27)
('Acaso', 28)
('Acción', 29)
('Aceptación', 30)
('Aceptó', 31)
('Achaguas', 32)
('Acometido', 33)
('Acupe', 34)
('Acusación', 35)
('Acuérdese', 36)
('Acuéstate', 37)
('Acábese', 38)
('Adelante', 39)
('Además', 40)
('Adoración', 41)
('Advierte', 42)
('Afortunadamente', 43)
('Afuera', 44)
('Agazapados', 45)
('Agradable', 46)
('Aguaita', 47)
('Aguaite', 48)
('Aguaiten', 49)
('Aguarde', 50)
```

Ahora lo ponemos todo junto en una clase tokenizadora

[16]:
```python
class TokenizadorSimpleV1:
    def __init__(self, vocabulario):
        self.str_to_int = vocabulario
        self.int_to_str = {i:s for s,i in vocabulario.items()}

    def encode(self, texto):
        preprocesado = re.split(r'([,.:;¿?_!¡"()\']|-|\s)', texto)

        preprocesado = [
            item.strip() for item in preprocesado if item.strip()
        ]
        ids = [self.str_to_int[s] for s in preprocesado]
        return ids

    def decode(self, ids):
        texto = " ".join([self.int_to_str[i] for i in ids])
        # Reemplazar los espacios antes de las puntuaciones especificadas
        texto = re.sub(r'\s+([,.?!"()\'])', r'\1', texto)
        return texto
```

- La función de codificación `encode` convierte el texto en identificadores de token
- La función de decodificación `decode` convierte los identificadores de token nuevamente en texto

[19]:
```
tokenizador = TokenizadorSimpleV1(vocabulario)

texto = 'Dos bogas lo hacen avanzar mediante una lenta y penosa maniobra de␣
 ↪galeotes.'
ids = tokenizador.encode(texto)
print(ids)
```

```
[464, 3097, 8723, 7498, 2820, 9086, 13592, 8512, 14026, 10271, 8920, 4847, 7202,
6]
```

- Podemos decodificar los números enteros y convertirlos en texto.

[20]:
```
tokenizador.decode(ids)
```

[20]:
```
'Dos bogas lo hacen avanzar mediante una lenta y penosa maniobra de galeotes.'
```

0.4.2 Agregar tokens de contexto especiales

- Es útil agregar algunos tokens "especiales" para palabras desconocidas y para indicar el final de un texto.
- Algunos tokenizadores utilizan tokens especiales para ayudar al LLM con contexto adicional
- Algunos de estos tokens especiales son
  - [BOS] (inicio de secuencia) marca el comienzo del texto
  - [EOS] (fin de secuencia) marca donde termina el texto (esto se usa generalmente para concatenar múltiples textos no relacionados, p. ej., dos artículos de Wikipedia diferentes o dos libros diferentes, etc.)
  - [PAD] (relleno) si entrenamos LLM con un tamaño de lote mayor a 1 (podemos incluir múltiples textos con diferentes longitudes; con el token de relleno, rellenamos los textos más cortos hasta la longitud más larga para que todos los textos tengan la misma longitud)
  - [UNK] para representar palabras que no están incluidas en el vocabulario
  - <|endoftext|> es análogo al token [EOS] mencionado anteriormente
- Utilizamos los tokens <|endoftext|> entre dos fuentes de texto independientes

[21]:
```
texto = "Hola, ¿te gusta el té? ¿Es esto una prueba?"
tokenizador.encode(texto)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[21], line 2
      1 texto = "Hola, ¿te gusta el té? ¿Es esto una prueba?"
----> 2 tokenizador.encode(texto)

Cell In[16], line 12, in TokenizadorSimpleV1.encode(self, texto)
      7 preprocesado = re.split(r'([,.:;¿?_!¡"()\']|-|\s)', texto)
```

```
      9 preprocesado = [
     10     item.strip() for item in preprocesado if item.strip()
     11 ]
---> 12 ids = [self.str_to_int[s] for s in preprocesado]
     13 return ids

Cell In[16], line 12, in <listcomp>(.0)
      7 preprocesado = re.split(r'([,.:;¿?_!¡"()\']|-|\s)', texto)
      9 preprocesado = [
     10     item.strip() for item in preprocesado if item.strip()
     11 ]
---> 12 ids = [self.str_to_int[s] for s in preprocesado]
     13 return ids

KeyError: 'Hola'
```

- Lo anterior produce un error porque la palabra "Hola" no está incluida en el vocabulario.
- Para solucionar estos casos, podemos agregar tokens especiales como "<|unk|>" al vocabulario para representar palabras desconocidas.
- Como ya estamos ampliando el vocabulario, agreguemos otro token llamado "<|endoftext|>"

[22]:
```
todos_los_tokens = sorted(list(set(preprocesado)))
todos_los_tokens.extend(["<|endoftext|>", "<|unk|>"])

vocabulario = {token:entero for entero,token in enumerate(todos_los_tokens)}
```

[23]:
```
len(vocabulario.items())
```

[23]: 14726

[24]:
```
for i, item in enumerate(list(vocabulario.items())[-5:]):
    print(item)
```

```
('"-Yo', 14721)
('"-Ése', 14722)
('"', 14723)
('<|endoftext|>', 14724)
('<|unk|>', 14725)
```

- También debemos ajustar el tokenizador en consecuencia para que sepa cuándo y cómo usar el nuevo token <unk>

[25]:
```
class TokenizadorSimpleV2:
    def __init__(self, vocabulario):
        self.str_to_int = vocabulario
        self.int_to_str = {i:s for s,i in vocabulario.items()}

    def encode(self, texto):
```

```
        preprocesado = re.split(r'([,.:;¿?_!¡"()\']|-|\s)', texto)

        preprocesado = [
            item.strip() for item in preprocesado if item.strip()
        ]
        preprocesado = [
            item if item in self.str_to_int
            else "<|unk|>" for item in preprocesado
        ]
        ids = [self.str_to_int[s] for s in preprocesado]
        return ids

    def decode(self, ids):
        texto = " ".join([self.int_to_str[i] for i in ids])
        # Reemplazar los espacios antes de las puntuaciones especificadas
        texto = re.sub(r'\s+([,.?!"()\'])', r'\1', texto)
        return texto
```

[29]:
```
tokenizador2 = TokenizadorSimpleV2(vocabulario)
texto = "Hola, ¿te gusta el té? ¿Es esto una prueba?"
tokenizador2.encode(texto)
```

[29]:
```
[14725,
 4,
 14087,
 13019,
 7411,
 5836,
 14725,
 15,
 14087,
 536,
 6605,
 13592,
 11054,
 15]
```

[31]:
```
texto1 = "Hola, ¿te gusta el té?"
texto2 = "En las soleadas terrazas del palacio."

texto = " <|endoftext|> ".join((texto1, texto2))

print(texto)
```

```
Hola, ¿te gusta el té? <|endoftext|> En las soleadas terrazas del palacio.
```

[32]:
```
tokenizador2.encode(texto)
```

[32]:
```
[14725,
 4,
 14087,
 13019,
 7411,
 5836,
 14725,
 15,
 14724,
 509,
 8457,
 14725,
 14725,
 4980,
 14725,
 6]
```

[33]:
```
tokenizador2.decode(tokenizador2.encode(texto))
```

[33]:
```
'<|unk|>, ¿ te gusta el <|unk|>? <|endoftext|> En las <|unk|> <|unk|> del
<|unk|>.'
```

0.4.3 Tokenizador usando el algoritmo BytePair

- El tokenizador BPE original se puede encontrar aquí: https://github.com/openai/gpt-2/blob/master/src/encoder.py
- Utilizamos el tokenizador BPE de la biblioteca de código abierto tiktoken de OpenAI, que implementa sus algoritmos centrales en Rust para mejorar el rendimiento computacional.

[34]:
```
import importlib
import tiktoken

print("Versión de tiktoken:", importlib.metadata.version("tiktoken"))
```

```
Versión de tiktoken: 0.8.0
```

[35]:
```
tokenizador3 = tiktoken.get_encoding("gpt2")
```

[38]:
```
texto1 = "Hola, ¿te gusta el té?"
texto2 = "En las soleadas terrazas del palacio."

texto = " <|endoftext|> ".join((texto1, texto2))

enteros = tokenizador3.encode(texto, allowed_special={"<|endoftext|>"})

print(enteros)
```

```
[39, 5708, 11, 1587, 123, 660, 35253, 64, 1288, 256, 2634, 30, 220, 50256, 2039,
39990, 6195, 38768, 1059, 3247, 292, 1619, 6340, 48711, 13]
```